Abstract

Compared to other arithmetic operations, division is not simple or quick in hardware. Division typically requires significantly more hardware to implement when compared to other arithmetic operations. In this lab, we compared four different implementations of hardware dividers on the FPGA to consider how many combinational logic elements are required to implement each. We compared a Qsys generated design and three designs that we manually configured: non-restoring, restoring, and long division dividers. For each divider, we made three versions of bit width four, eight, and sixteen to give us perspective on how each scales in space. We compared these three designs in the relative resource usage when loaded onto the FPGA. For each division algorithm, we tested the correctness of their outputs using the hex display on the Altera board.

Hardware Division

Hardware division is nontrivial and takes a relatively larger amount of gates to implement when compared to other basic math functions. Each divider was implemented in three bit widths: four, eight, and sixteen. Three of our dividers were implemented in VHDL, and one was implemented using System Verilog. While the Verilog implementation was written directly, the VHDL implementations we configured using block diagrams in HDL designer and then used the computer-generated VHDL.

The System Verilog divider was implemented by taking the instructor provided code and simply changing it to do division. When importing the VHDL code we generated we also used that code as a harness for our own implementations which allowed us to make use of the 7-segment display on the Altera board. The code for our Verilog implementation is given below and is also available on the wiki page.

Figure 1: Verilog used to generate divider

```
module Divider(CLOCK_50, A_set, B_set, Reset, Switches, HexOut);
input A_set; // Click to set the value of A
input B_set; // Click to set the value of B
input Reset; // Sets A and B to 0
input CLOCK_50; // Input clock, 50 MHz
input [7:0] Switches; // Switches 0 through 7
output [27:0] HexOut; // Hex display signals

reg [7:0] A; // Stored value of A
reg [7:0] B; // Stored value of B
wire [15:0] QuotientOut; // Input signal to hex driver, quotient of
A*B

always @(posedge CLOCK_50)
```

```
begin
      if (Reset == 0) // Reset condition (pushbuttons are active-low)
            begin
                  A <= 8'b0;
                  B <= 8'b0;
            end
      else
            begin
                  if (A_set == 0) // Setting A has the priority
                        A <= Switches;
                  else if (B_set == 0) // set B
                        B <= Switches;
            end
end

Hex_Driver hd0 (.In0(QuotientOut[3:0]), .Out0(HexOut[6:0]));

Hex_Driver hd1 (.In0(QuotientOut[7:4]), .Out0(HexOut[13:7]));

Hex_Driver hd2 (.In0(QuotientOut[11:8]), .Out0(HexOut[20:14]));

Hex_Driver hd3 (.In0(QuotientOut[15:12]), .Out0(HexOut[27:21]));


assign QuotientOut = B/A;

endmodule
```

The first hardware implementation that we constructed manually uses the non-restoring division algorithm. This algorithm involves iterating over the bits and computing a partial remainder at every step dependent on the sign of the partial remainder on the last iteration. In hardware, we implemented this by using a comparator, multiplier, adder, subtractor, and two multiplexers. We compute two partial remainders in parallel and select the one we want to pass on to the next stage by using the comparator/multiplexer configuration. Our design for one stage of the non-restoring divider can be seen to the right in the form of a block diagram constructed in HDL designer. The algorithm for this type of division is shown also below. It is important to note that there is also some additional logic at the beginning and end of this algorithm. Before any partial remainders can be computed, both the inputted numerator and denominator must be extended to contain double the initial bit width; additionally, the denominator must be shifted by a value equal to the original number of bits. At the end, the quotient is subtracted

by its own bit complement to produce the final quotient output.

Figure 2: Non-restoring divider stage

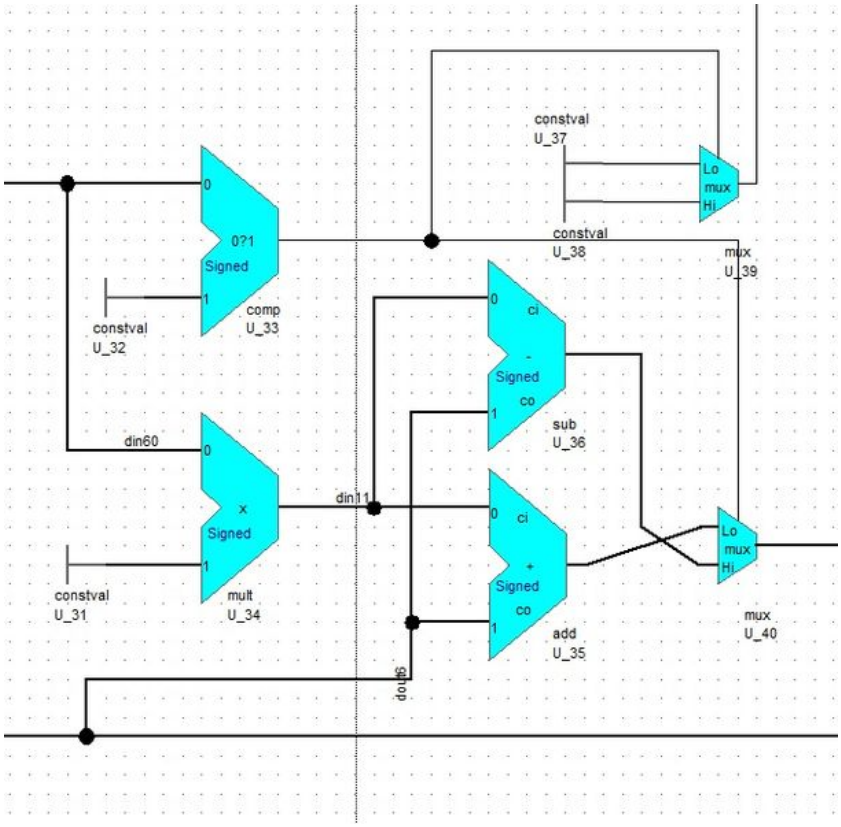


Figure 3: Non-restoring division algorithm[1]

```
P := N
 D := D << n             * P and D need twice the word width of N and
Q
 for i = n-1..0 do       * for example 31..0 for 32 bits
   if P >= 0 then
     q[i] := +1
     P := 2*P - D
```

```
    else
      q[i] := -1
      P := 2*P + D
    end if
  end
* Note: N=Numerator, D=Denominator, n=#bits, P=Partial remainder,
q(i)=bit #i of quotient.
```

The second hardware division implementation that we considered uses the restoring division algorithm. While non-restoring division does not restore the partial remainders it computes at each step, the restoring division algorithm does. To implement this in hardware we made use of a similar set of hardware as we did with the non-restoring divider, but in a different configuration. In this case, the divider does a "trial subtraction" where it computes the partial remainder, then if the subtraction was successful and the partial remainder is non-negative it sets the bit high, or else it sets the bit low and restores the partial remainder. Our design for one stage of the restoring divider can be seen below along with the restoring division algorithm. Note that, like the non-restoring version, this algorithm requires that the inputted numerator and denominator are extended to double the bit width, and the denominator is then shifted by a value equal to the original number of bits.

Figure 4: Restoring divider stage

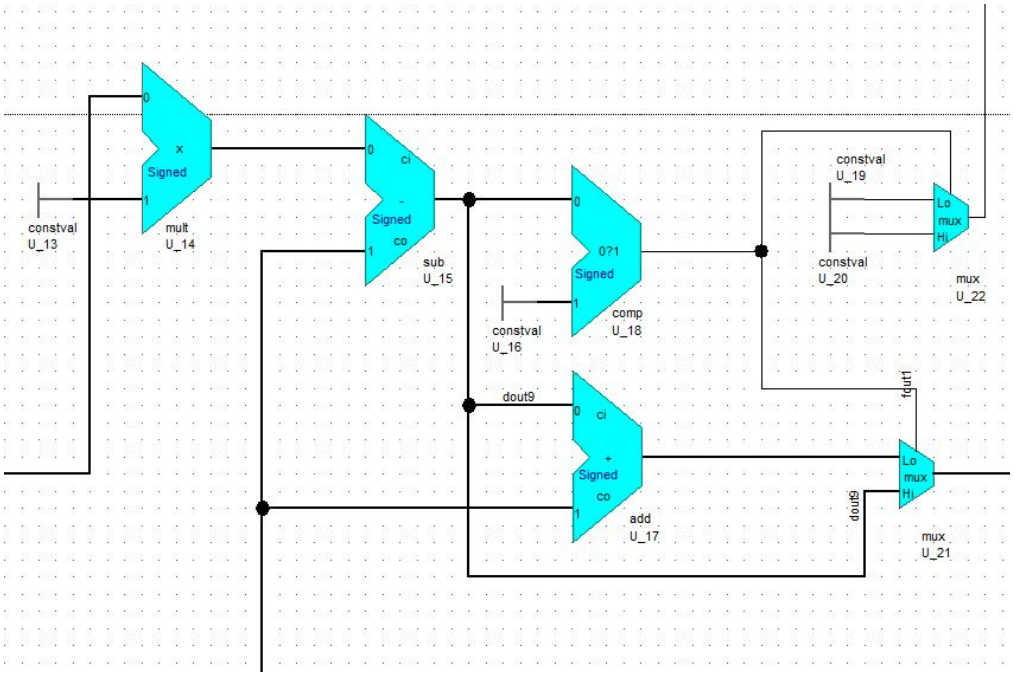


Figure 5: Restoring divider algorithm[2]

```
P := N
D := D << n                -- P and D need twice the word width of N and
Q
for i = n-1..0 do          -- for example 31..0 for 32 bits
  P := 2P - D              -- trial subtraction from shifted value
  if P >= 0 then
    q(i) := 1              -- result-bit 1
  else
    q(i) := 0              -- result-bit 0
    P := P + D             -- new partial remainder is (restored)
  end                         shifted value
end

-- Where: N = Numerator, D = Denominator, n = #bits, P = Partial
remainder, q(i) = bit #i of quotient
```

The final implementation we made in VHDL was the long division divider. This divider takes the traditional long division approach in implementing binary division.

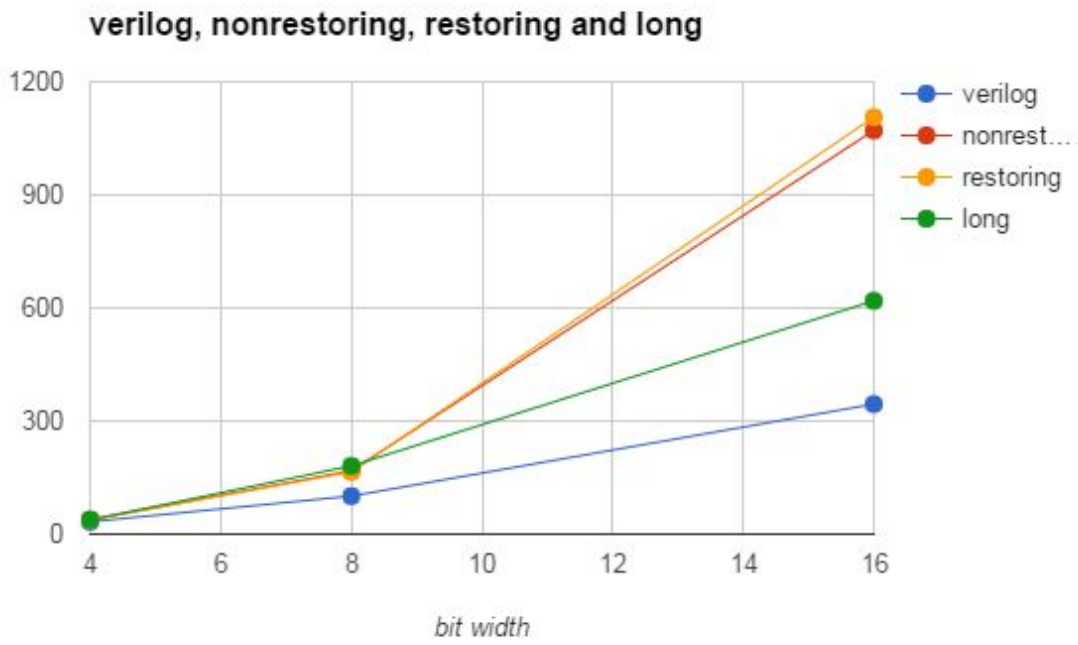
Figure 6: Long division divider stage

Figure 7: Long division algorithm

```
Q := 0                  -- initialize quotient and remainder to zero
R := 0
for i = n-1...0 do      -- where n is number of bits in N
  R := R << 1           -- left-shift R by 1 bit
  R(0) := N(i)          -- set the least-significant bit of R equal to
  if R >= D then           bit i of the numerator
    R := R - D
    Q(i) := 1
  end
end
```

From our results we saw that the Systemverilog implementation of division was the most successful implementation as it took a significantly smaller amount of logical elements to implement. On average it was around 40% to 50% smaller than the VHDL implementations of the division algorithms for the 8 bit implementations and . This might be due to the fact that the restoring and non restoring algorithms were meant to be used with a clock and the long division algorithm is an emulation of what is done by hand to do division. Therefore none of our three VHDL implementations were optimized compared to the systemVerilog inferred implementation of a divider circuit. The space scaling of each circuit is does not scale the same. The systemVerilog implementation scales much better than the other implementations. The long division beats both of the restoring and non restoring division hardware implementation/ This leads to the conclusion that our implementations are just not as optimized as the Systemverilog implementation and that will show dramatically as we scale the bit widths for these dividers upwards. The scaling of

each divider looks nonlinear but the order of scaling is unclear to us. Please refer to the figure below for the comparisons on how well each of the dividers scale.



All three of these dividers started out with comparable sizes (actually plus minus one or two) around 35. They then scale upwards and start to differentiate and differentiate in terms of size. We can safely say that each one of these implementations scale differently with bit width. These differences could lie within multiple variables including how each algorithm works as well as how these algorithms were actually implemented in synthesizable hardware.

Sources

1. https://en.wikipedia.org/wiki/Division_algorithm#Non-restoring_division
2. https://en.wikipedia.org/wiki/Division_algorithm#Restoring_division
3. https://en.wikipedia.org/wiki/Division_algorithm#Long_division